ALGORITHMIC ASPECTS OF NATURAL LANGUAGE PROCESSING

Mark-Jan Nederhof, University of St Andrews
Giorgio Satta, University of Padua

1 Introduction

Examples of natural languages are Chinese, English and Italian. They are called *natural* as they evolved in a more or less natural way, without too many deliberate considerations. This sets them apart from **formal languages**, amongst which are programming languages, which are designed to allow easy processing by computer algorithms. Typically, programs in programming languages such as C or Java can be processed (compiled) in close to linear time in their length.

One particular feature that most programming languages have in common, and that allows for their fast processing, is absence of **ambiguity**. That is, only one structure, called a **parse** or **parse tree**, can be assigned to any program, and this parse can have only one meaning. Furthermore, the design of many programming languages is such that the single parse can be found **deterministically**, which means that every parsing step contributes a fragment of the resulting parse. As parses have a size linear in the length of the input, this explains why parsing is possible in linear time. Subsequent processing of the parse, for example in order to compile to machine code, is also commonly possible in close to linear time.

Natural languages are quite different in this respect. Like programs in a programming language, sentences in a natural language can be assigned parses, but often the sentences are ambiguous and allow more than one parse. Even for a single parse, there may be ambiguity in the meanings of words or expressions. The existence of ambiguity in natural language is witnessed by frequent misunderstandings in daily life, but it is also an essential feature of poetry and puns.

The field of **natural language processing** (NLP) studies algorithms, tools and techniques for automatic processing of natural languages. A related if not synonymous term is **computational linguistics**, which stresses that the field can be seen as a subfield of **linguistics**, which is the study of (natural) language.

Language can be investigated from different perspectives. At the lowest level, **phonetics** and **phonology** study the sounds that spoken language consists of and the rules that govern them. **Morphology** is the study of the internal structure of words. How words are combined to form sentences is the subject of **syntax**. The meaning and use of language are studied by **semantics** and **pragmatics**, and **discourse** studies the structure of human communication.

The problem of parsing, with which we started our exposition, concerns the syntax of language. Although interesting algorithms exist for the other levels of language as well, this chapter will concentrate on algorithms related to syntax, as these form the most mature, and well-understood part of NLP.

Next to parsing, many other tasks studied in NLP are relevant to syntax. One such task is **grammar induction**, which involves finding the syntactic structure of a language on the basis of examples. Another task is **machine translation**, which includes transforming syntactic structure from one language to that of another. A selection of algorithms that concern these tasks will be discussed further in the remainder of this chapter.

2 Example

We illustrate the problem of ambiguity by an example. Familiarity with context-free grammars (Chapter **XX** [Formal Grammars and Languages]) is assumed.

The sentence:

(1) our company is training workers

has one obvious meaning to most readers. However, at least two other meanings exist in principle. This becomes clear if we replace some words by other words of the same type:

(2) our *problem* is training workers

(3) our company *hires* training workers

The three readings of (1) correspond to the three parses in Figure 1, assuming the context-free grammar (CFG) below. This grammar is intended for

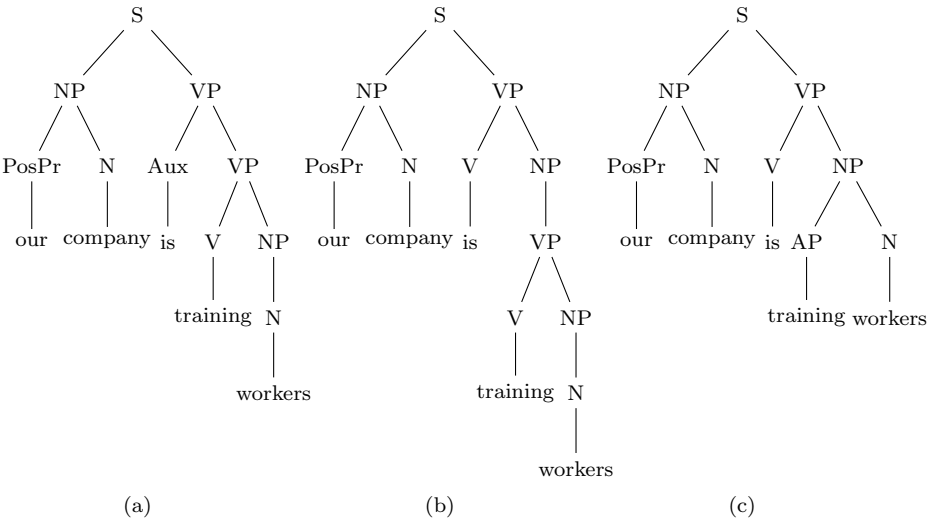

Figure 1: Three parses for 'our company is training workers'.

illustrative purposes, and represents a rough approximation of a tiny part of English at best. We let S stand for 'sentence', NP for 'noun phrase', N for 'noun', PosPr for 'possessive pronoun', VP for 'verb phrase', Aux for 'auxiliary verb', V for 'verb', and AP for 'adjective phrase'.

S	→	NP VP
NP	→	N
NP	→	PosPr N
NP	→	AP N
NP	→	VP
N	→	company
N	→	workers
PosPr	→	our
VP	→	Aux VP
VP	→	V NP
Aux	→	is
V	→	training
V	→	is
AP	→	training

An important observation is that the number of parses can be exponential in the length of the sentence. An easy way to illustrate this is to extend

the above example by conjunction, to allow sentences of the form:

(4) Robin's company is training workers and Sandy's company is training workers and [...]

Such a sentence has at least 3^k parses, where k is the number of times the ambiguous construction from (1) is repeated.

The exponential behaviour entails that in practice it is not feasible to enumerate all parses of a sentence. Therefore one often represents the set of all parses by a structure commonly referred to as a **parse forest**, also called more explicitly a **shared-packed parse forest**. The term 'parse forest' has become popular since the book by Tomita (1986), but the underlying concept existed long before that, as we will show in the next section.

Two observations underlie parse forests. The first is that alternative subparses of a substring with the same nonterminal at the root can be 'packed' together, and be treated as one with regard to larger subparses of larger substrings. In Figure 1, this pertains to the two subparses for 'training workers' with NP at the root in parses (b) and (c). The second observation is that an identical subparse can be 'shared' among several larger subparses. An example in Figure 1 is the subparse of 'training workers' with root labelled VP, which is the same in parses (a) and (b).

A graphical representation of a parse forest is given in Figure 2. Packing is represented by a rectangle enclosing two or more nodes. We see sharing where a node has more than one parent.

In the following section, we discuss construction of parse forests. Thereafter, we also address the problem of selecting the intended parse tree within a parse forest.

3 Context-free parsing by intersection

The underlying principle of parse forests was already discovered by Bar-Hillel et al. (1964), who found a constructive proof that the intersection of a context-free language and a regular language is again a context-free language. The input to this construction is a context-free grammar and a finite automaton. Instead of a graph representation as in Figure 2, the output is itself a context-free grammar, which is called the **intersection grammar**. In this representation, sharing amounts to multiple occurrences of a nonterminal in right-hand sides, and packing amounts to multiple occurrences of a nonterminal in left-hand sides.

We simplify the discussion here by assuming a special type of finite automaton that recognizes just one string, say $w = a_1 \cdots a_n$. The states

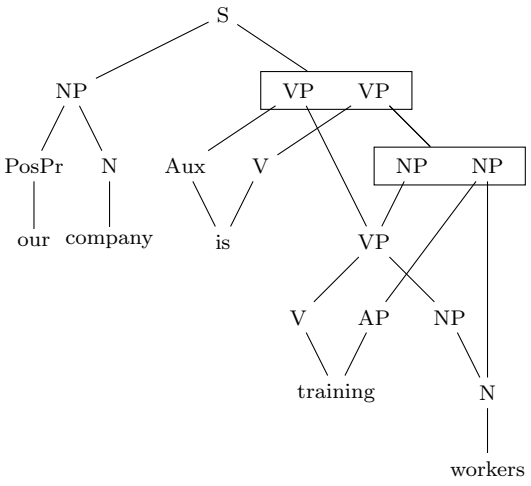

Figure 2: The three parses in a parse forest.

of the automaton represent the positions between adjacent symbols. In addition, the initial state 0 and the final state n represent the positions at the left and right ends of w. By this simplification, nonterminals of the intersection grammar are triples $[i, X, j]$, where i and j are input positions, and X is a terminal or nonterminal from the input grammar. For each subparse with root labelled $[i, X, j]$ $(0 \leq i \leq j \leq n)$ in the intersection grammar, there is a corresponding subparse of $a_{i+1} \cdots a_j$ with root labelled X in the original grammar. Triples of this form, as well as related objects to be discussed in the sequel, are also referred to as (parse) **items**. The start symbol of the intersection grammar is $[0, S, n]$, where S is the start symbol of the input grammar.

As an example, Figure 3 shows the parse forest from Figure 2 in the representation as intersection grammar.

Let $\mathcal{G}$ be the input CFG and w the input string of length $|w| = n$. An easy way to construct the intersection grammar $\mathcal{G}_\cap$ is by the algorithm below.

INTERSECTION$(\mathcal{G}, w)$ {let $a_1 \cdots a_n = w$}
1: $\mathcal{G}_\cap \leftarrow$ CFG with start symbol $[0, \mathrm{S}, n]$ and empty set of rules
2: **for all** rules $A \rightarrow X_1 \cdots X_m$ from $\mathcal{G}$ **do**
3: **for all** sequences of positions $i_0, \ldots, i_m$ $(0 \leq i_0 \leq \ldots \leq i_m \leq n)$ **do**
4: add the rule $[i_0, A, i_m] \rightarrow [i_0, X_1, i_1] \cdots [i_{m-1}, X_m, i_m]$ to $\mathcal{G}_\cap$
5: **for all** i $(1 \leq i \leq n)$ **do**

$$
\begin{array}{lcl}
[0, \mathrm{S}, 5] & \to & [0, \mathrm{NP}, 2]\ [2, \mathrm{VP}, 5] \\
[0, \mathrm{NP}, 2] & \to & [0, \mathrm{PosPr}, 1]\ [1, \mathrm{N}, 2] \\
[0, \mathrm{PosPr}, 1] & \to & [0, \mathrm{our}, 1] \\
[1, \mathrm{N}, 2] & \to & [1, \mathrm{company}, 2] \\
[2, \mathrm{VP}, 5] & \to & [2, \mathrm{Aux}, 3]\ [3, \mathrm{VP}, 5] \\
[2, \mathrm{VP}, 5] & \to & [2, \mathrm{V}, 3]\ [3, \mathrm{NP}, 5] \\
[2, \mathrm{Aux}, 3] & \to & [2, \mathrm{is}, 3] \\
[2, \mathrm{V}, 3] & \to & [2, \mathrm{is}, 3] \\
[3, \mathrm{NP}, 5] & \to & [3, \mathrm{VP}, 5] \\
[3, \mathrm{NP}, 5] & \to & [3, \mathrm{AP}, 4]\ [4, \mathrm{N}, 5] \\
[3, \mathrm{VP}, 5] & \to & [3, \mathrm{V}, 4]\ [4, \mathrm{NP}, 5] \\
[3, \mathrm{V}, 4] & \to & [3, \mathrm{training}, 4] \\
[3, \mathrm{AP}, 4] & \to & [3, \mathrm{training}, 4] \\
[4, \mathrm{NP}, 5] & \to & [4, \mathrm{N}, 5] \\
[4, \mathrm{N}, 5] & \to & [4, \mathrm{workers}, 5]
\end{array}
\qquad
\begin{array}{lcl}
[0, \mathrm{our}, 1] & \to & \mathrm{our} \\
[1, \mathrm{company}, 2] & \to & \mathrm{company} \\
[2, \mathrm{is}, 3] & \to & \mathrm{is} \\
[3, \mathrm{training}, 4] & \to & \mathrm{training} \\
[4, \mathrm{workers}, 5] & \to & \mathrm{workers}
\end{array}
$$

Figure 3: Parse forest in the representation as intersection grammar.

6: add the rule $[i-1, a_i, i] \to a_i$ to $\mathcal{G}_\cap$
7: **return** $\mathcal{G}_\cap$

In general, the above algorithm produces more rules than needed. Formally, we say a nonterminal in a CFG is **generating** if at least one terminal string can be derived from that nonterminal. We say a nonterminal is **reachable** if a string containing that nonterminal can be derived from the start symbol. A nonterminal is called **useless** if it is non-generating or unreachable or both. A grammar is called **reduced** if it does not contain any rules with useless nonterminals.

In the running example, $[0, \mathrm{VP}, 2]$ is non-generating, as 'our company' is not a verb phrase. Therefore, the presence of a rule $[0, \mathrm{VP}, 2] \to [0, \mathrm{Aux}, 1]\ [1, \mathrm{VP}, 2]$ makes the intersection grammar non-reduced.

The set of generating nonterminals can be computed by the algorithm below. It is applicable to any CFG, not just to those with nonterminals of the form $[i, A, j]$. We let Σ denote the terminal alphabet, and Σ^* the set of strings of terminal symbols, including the empty string ε.

GENERATING($\mathcal{G}$)
1: $OLDGEN \leftarrow \emptyset$
2: $GEN \leftarrow \{A \mid A \to v, v \in \Sigma^*\}$
3: **while** $GEN \neq OLDGEN$ **do**
4: $\quad OLDGEN \leftarrow GEN$
5: $\quad GEN \leftarrow \{A \mid A \to \alpha, \alpha \in (\Sigma \cup OLDGEN)^*\}$
6: **return** GEN

This algorithm can be implemented to run in linear time in the number

of nonterminal occurrences in $\mathcal{G}$, in the following way. For each rule, we maintain a counter containing the number of nonterminals in the right-hand side that have not yet been added to *GEN*. If a nonterminal A is newly added to *GEN*, the counters of rules in which A occurs in the right-hand side are decremented. If a counter of any rule becomes 0, the left-hand side nonterminal is added to *GEN*, unless it has already been added. By this implementation, at most one step is needed for each nonterminal occurrence in the grammar.

Also the problem of finding the set of reachable nonterminals can be solved in linear time, by simple reduction to graph-reachability. We conclude that a grammar can be reduced in linear time, by computing the generating and reachable nonterminals, and then eliminating all rules in which nonterminals occur that are not generating or not reachable.

In practice it is preferable to combine the construction of an intersection grammar with its reduction, which diminishes the space requirements of intermediate results. The procedure starts by computing a set of generating nonterminals. The difference with the procedure above is that this phase now precedes the explicit construction of rules of the intersection grammar.

The algorithm does however store and return items of the form $[i, A \to \alpha \bullet \beta, j]$, where $A \to \alpha\beta$ is a rule from $\mathcal{G}$ and $\alpha \neq \varepsilon$. Informally, the dot divides the right-hand side into a (non-empty) prefix and a suffix, and the input positions i and j delimit a substring that can be derived from the prefix. More precisely, if we assume $\alpha = X_1 \cdots X_m$, then an item of this form represents that $[i_0, X_1, i_1]$, $\ldots$, $[i_{m-1}, X_m, i_m]$ are all generating nonterminals, for some choice of $i = i_0$, i_1, $\ldots$, i_{m-1}, $i_m = j$. An item $[i, A \to \alpha \bullet \beta, j]$ can thereby be seen as a partial result towards establishing that $[i, A, j']$ may be a generating nonterminal, for some $j' \geq j$. Such items together with the familiar items of the form $[i, A, j]$ are gathered in a single set *GEN*.

The algorithm maintains an agenda with newly obtained items that are still to be put in *GEN*. New items are also combined with existing items to derive yet more items, until the agenda is empty.

GeneratingIntersection$(\mathcal{G}, w)$ {let $a_1 \cdots a_n = w$}

1: $GEN \leftarrow \emptyset$
2: $AGENDA \leftarrow \{[i-1, a_i, i] \mid 1 \leq i \leq n\} \cup \{[i, A, i] \mid A \to \varepsilon, 0 \leq i \leq n\}$
3: **while** $AGENDA \neq \emptyset$ **do**
4: remove some $ITEM$ from $AGENDA$
5: **if** $ITEM \notin GEN$ **then**

```
      GEN ← GEN ∪ {ITEM}
      if ITEM = [j, X, k] then
        for all [i, A → α • Xβ, j] ∈ GEN do
          AGENDA ← AGENDA ∪ {[i, A → αX • β, k]}
        for all A → Xβ do
          AGENDA ← AGENDA ∪ {[j, A → X • β, k]}
      if ITEM = [i, A → α • Xβ, j] then
        for all [j, X, k] ∈ GEN do
          AGENDA ← AGENDA ∪ {[i, A → αX • β, k]}
      if ITEM = [i, A → α •, j] then
        GEN ← GEN ∪ {[i, A, j]}
return GEN
```

The rules of the reduced intersection grammar are now constructed in a top-down manner, which ensures that only reachable nonterminals are considered. The process is initiated by a call IntersectionFiltered$(0, S, n)$, where S is the start symbol of the input grammar and $n = |w|$. A set *DONE*, which is initially $\emptyset$, is maintained to prevent that rules would be constructed more than once by repeated calls of IntersectionFiltered with the same arguments. All new rules are composed of nonterminals that were found to be generating earlier. This guarantees that the intersection grammar is reduced.

IntersectionFiltered(i, X, j)

```
if [i, X, j] ∉ DONE then
  DONE ← DONE ∪ {[i, X, j]}
  if X is terminal a then
    add [i, a, j] → a to G∩
  else {X is nonterminal A}
    if i = j and A → ε then
      add [i, A, j] → ε to G∩
    for all A → X1 ··· Xm (m > 0) and sequences
      [i0, A → X1 ··· Xm−1Xm •, im], [im−1, Xm, im],
      [i0, A → X1 ··· Xm−1 • Xm, im−1], [im−2, Xm−1, im−1],
      ...,
      [i0, A → X1 • ··· Xm−1Xm, i1], [i0, X1, i1] ∈ GEN,
    where i0 = i and im = j do
      add [i0, A, im] → [i0, X1, i1] ··· [im−1, Xm, im] to G∩
      for all k (1 ≤ k ≤ m) do
        IntersectionFiltered(ik−1, Xk, ik)
```

If r is the length of the longest right-hand side of a rule from the in-

put grammar $\mathcal{G}$, then the size $|\mathcal{G}_\cap|$ of $\mathcal{G}_\cap$ is $\mathcal{O}(|\mathcal{G}| \cdot n^{r+1})$. The size of a context-free grammar is defined as the sum of the number of nonterminal and terminal occurrences in its rules. Also the time and space complexity of the intersection process itself are given by $\mathcal{O}(|\mathcal{G}| \cdot n^{r+1})$, irrespective of whether reduction is integrated into the intersection process, or done afterwards.

If the input grammar is in **binary form** (that is $r = 2$), then the complexity is $\mathcal{O}(|\mathcal{G}| \cdot n^3)$. This is also the time and space complexity of the best practical recognition and parsing algorithms for CFGs. (See for example the CYK algorithm discussed in Chapter **XX** [Formal Grammars and Languages].) Grammars that are not in binary form can be easily brought in binary form by a construction that is linear in the size of the input grammar.

It is also possible to integrate binarization into the construction of the intersection grammar, by using items of the form $[i, A \rightarrow \alpha \bullet \beta, j]$, $\alpha \neq \varepsilon$, as nonterminals. The intersection grammar then contains, amongst others, rules of the form $[i, A \rightarrow \alpha X \bullet \beta, k] \rightarrow [i, A \rightarrow \alpha \bullet X \beta, j]\ [j, X, k]$ and of the form $[i, A \rightarrow X \bullet \beta, j] \rightarrow [i, X, j]$.

An important observation is that GeneratingIntersection has time complexity $\mathcal{O}(|\mathcal{G}| \cdot n^3)$ but its space complexity is only $\mathcal{O}(|\mathcal{G}| \cdot n^2)$. If the objective is recognition rather than parsing, then it suffices to check whether $[0, S, n]$ is in *GEN* and application of IntersectionFiltered is not needed.

Extraction of a single and arbitrary parse from *GEN* has time complexity $\mathcal{O}(|\mathcal{G}| \cdot n^2)$. This can be explained as follows. There are $\mathcal{O}(|\mathcal{G}| \cdot n)$ items of the form $[i_0, A \rightarrow X_1 \cdots X_k \bullet \cdots X_m, i_k]$, $k \geq 2$, that are consistent with a given parse of the input string. Such items are considered while we are extracting the parse from *GEN* in a top-down manner, much as in IntersectionFiltered. For each, we may need to check at most $\mathcal{O}(n)$ values i_{k-1} between i_0 and i_k, to determine whether $[i_0, A \rightarrow X_1 \cdots \bullet X_k \cdots X_m, i_{k-1}]$, $[i_{k-1}, X_k, i_k] \in \mathit{GEN}$.

One may extend GeneratingIntersection such that it stores a list of the relevant i_{k-1} with items $[i_0, A \rightarrow X_1 \cdots X_k \bullet \cdots X_m, i_k]$. This allows a single parse to be extracted in $\mathcal{O}(|\mathcal{G}| \cdot n)$ time, but the storage costs then grow to $\mathcal{O}(|\mathcal{G}| \cdot n^3)$.

Algorithms such as GeneratingIntersection are sometimes presented in the form of a **deduction system**. A deduction system consists of a collection of **inference rules**, each consisting of a list of **antecedents**, which stand for items that we have already derived, and, below a horizontal line, the **consequent**, which stands for an item that we derive from the antecedents. At the right of an inference rule, we may also write a number

$$\frac{}{[i-1, a_i, i]} \{1 \leq i \leq n \quad \text{(a)} \qquad \frac{}{[i, A, i]} \begin{cases} A \to \varepsilon \\ 0 \leq i \leq n \end{cases} \quad \text{(b)}$$

$$\frac{[i, A \to \alpha \bullet, j]}{[i, A, j]} \quad \text{(c)} \qquad \frac{[j, X, k]}{[j, A \to X \bullet \beta, k]} \{A \to X\beta \quad \text{(d)}$$

$$\frac{[i, A \to \alpha \bullet X\beta, j] \quad [j, X, k]}{[i, A \to \alpha X \bullet \beta, k]} \quad \text{(e)}$$

Figure 4: The algorithm GeneratingIntersection as deduction system.

of **side conditions**, which need to be fulfilled for the inference rule to be applicable. The side conditions here refer to rules from the grammar and the input string.

A deduction system has a natural interpretation as dynamic programming algorithm, exemplified by our code for GeneratingIntersection, as realization of the deduction system in Figure 4. An easy way to determine the time complexity of the dynamic programming algorithm is to look at the number of possible instantiations of each inference rule. The inference rule in Figure 4(e) is the most expensive, as it involves three input positions and one position within a grammar rule. The number of applications is therefore $\mathcal{O}(|\mathcal{G}| \cdot n^3)$, which confirms our earlier observations about the time complexity of GeneratingIntersection.

4 Lexicalization

A central issue in modeling the syntax of natural languages is the extreme sensitivity to the choice of lexical elements, that is, the words of the language. Let us return to the example in Section 2. The appropriate parse of 'our company is training workers' is that of Figure 1(a). However, if we replace the word 'company' with the word 'problem', the appropriate parse becomes the structure reflected by Figure 1(b) instead. A CFG of the kind presented in Section 2 cannot model such effects, as it treats 'company' and 'problem' uniformly as nouns (strings derived from N).

A solution is to incorporate a terminal symbol (lexical element) in each nonterminal. This terminal is such that it plays an important role in the syntactic and semantic content of the derived string. The model we consider here is called **bilexical context-free grammar** (2-LCFG). In various guises, this model is used extensively in natural language parsing. It allows

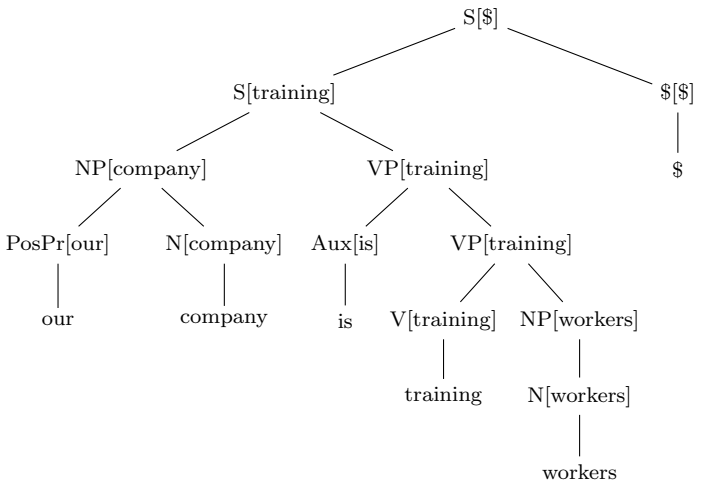

Figure 5: A parse of 'our company is training workers', assuming a bilexical context-free grammar.

us to write rules of the form S[training] → NP[company] VP[training], which expresses that a noun phrase whose main element is 'company' can combine with a verb phrase whose main element is 'training'. By the same token, one might exclude a rule of the form S[training] → NP[problem] VP[training], since, typically, a problem cannot be the subject of training.

More precisely, a 2-LCFG is a CFG with nonterminal symbols of the form $A[a]$, where a is a terminal symbol and A is a symbol drawn from a small set of so-called **delexicalized nonterminals**, which we denote as V_D. Every rule in a 2-LCFG has one of the following forms: $A[b] \rightarrow B[b]\ C[c]$, $A[c] \rightarrow B[b]\ C[c]$ or $A[a] \rightarrow a$. Note that in binary rules (those with two members in the right-hand side), the terminal symbol associated with the left-hand side nonterminal is always inherited from one of the nonterminals in the right-hand side.

For technical reasons, we assume existence of a dummy terminal \$ to the right of each sentence, and nowhere else. A dummy nonterminal with the same name allows terminal \$ to be derived by \$[\$] → \$. The start symbol of a 2-LCFG is $S[\$]$ and there are rules expanding it to $S[a]\ \$[\$]$, where a is typically the main verb of a sentence to be derived. Figure 5 presents a possible parse tree for the running example, assuming a 2-LCFG.

Let Σ denote the set of terminal symbols. In the worst case, a 2-LCFG can have $\Theta(|V_D|^3 \cdot |\Sigma|^2)$ binary rules. Whereas V_D is typically small, the

set Σ can grow very large in practical applications. When parsing with a 2-LCFG, it is therefore preferable to restrict the grammar to those rules that contain lexical elements actually occurring within the input sentence w. Based on the time complexity of general context-free parsing, as discussed in the previous sections, we then obtain a time complexity of $\mathcal{O}(|V_D|^3 \cdot |w|^5)$, under the assumption that $|w| < |\Sigma|$, which always holds in practical applications. In terms of sentence length, this is much worse than the time complexity of unlexicalized parsing.

The time complexity can be reduced by a factor of $|w|$ by a recognition algorithm that was specifically designed for lexicalized grammars. It uses items of the form $[i, A, h, j]$. In terms of the items used in Section 3, this has the same meaning as $[i, A[a_h], j]$, which now includes a lexicalized nonterminal $A[a_h]$. New is that some steps of the algorithm require temporarily ignoring either i or j, which is indicated by substituting one or the other by a hyphen.

We also need items of the form $[B, h, A, j, k]$ to indicate that $[-, B, h, j]$ and $[j, C, h', k]$ were derived, for some C and h' such that $A[a_h] \rightarrow B[a_h]$ $C[a_{h'}]$ is a rule. This represents an intermediate step in establishing $[i, A, h, k]$, temporarily ignoring the left boundary i of the substring derived from the left child $B[a_h]$, and forgetting the right child $C[a_{h'}]$. In a following step, i is reconstituted by access to an original item $[i, B, h, j]$.

Items of the form $[i, j, A, C, h]$ have a symmetrical meaning, that is, they indicate that $[i, B, h', j]$ and $[j, C, h, -]$ were derived, for some B and h' such that $A[a_h] \rightarrow B[a_{h'}]$ $C[a_h]$ is a rule.

The algorithm is given in Figure 6 as deduction system, and an illustration of the use of an important combination of inference rules is given in Figure 7. The deduction of $[i, A, h, k]$ from $[i, B, h, j]$ and $[j, C', h, k]$ via a grammar rule $A[a_h] \rightarrow B[a_h]$ $C[a_{h'}]$ is very similar to a step of the CYK algorithm for general context-free grammars. The current algorithm does the same in three different steps, represented by inference rules (e), (b) and (f). Each of these involves no more than four input positions and three delexicalized nonterminals. This corresponds to $\mathcal{O}(|V_D|^3 \cdot n^4)$ applications of each inference rule, which is also the total time complexity of the algorithm.

5 Probabilistic parsing

In natural language systems, parsing is commonly one stage of processing amongst several others. The effectiveness of the stages that follow parsing generally relies on having obtained a small set of preferred parses, ideally

$$\frac{}{[h-1, A, h, h]} \begin{cases} A[a_h] \rightarrow a_h \\ 1 \leq h \leq n+1 \end{cases} \quad \text{(a)}$$

$$\frac{[-, B, h, j] \quad [j, C, h', k]}{[B, h, A, j, k]} \left\{ A[a_h] \rightarrow B[a_h]\ C[a_{h'}] \right. \quad \text{(b)}$$

$$\frac{[i, B, h', j] \quad [j, C, h, -]}{[i, j, A, C, h]} \left\{ A[a_h] \rightarrow B[a_{h'}]\ C[a_h] \right. \quad \text{(c)}$$

$$\frac{[i, A, h, j]}{[i, A, h, -]} \quad \text{(d)}$$

$$\frac{[i, A, h, j]}{[-, A, h, j]} \quad \text{(e)}$$

$$\frac{[i, B, h, j] \quad [B, h, A, j, k]}{[i, A, h, k]} \quad \text{(f)}$$

$$\frac{[j, C, h, k] \quad [i, j, A, C, h]}{[i, A, h, k]} \quad \text{(g)}$$

Figure 6: Deduction system for bilexical recognition. We assume $w = a_1 \cdots a_n$, $a_{n+1} = \$$.

only one, from amongst the full set of parses, represented as a parse forest or intersection grammar. This is called (syntactic) **disambiguation**. There are roughly two ways to achieve this. First, some kind of filter may be applied to the full set of parses, to reject all but a few. This filter may look at the meanings of words and phrases, for example, and may be based on linguistic knowledge that is very different in character from the grammar that was used for parsing.

A second approach is to augment the parsing process so that probabilities are attached to parses and subparses. The higher the probability of a parse or subparse, the more confident we are that it is correct. This is called **probabilistic parsing**. The simplest form of probabilistic parsing relies on an assignment of probabilities to individual rules from a context-free grammar. These probabilities are then multiplied upon combination of rules to form parses.

As an example, consider the following probabilistic context-free grammar, which extends the grammar from the running example with the probabilities between parentheses. As before, the example is meant to illustrate

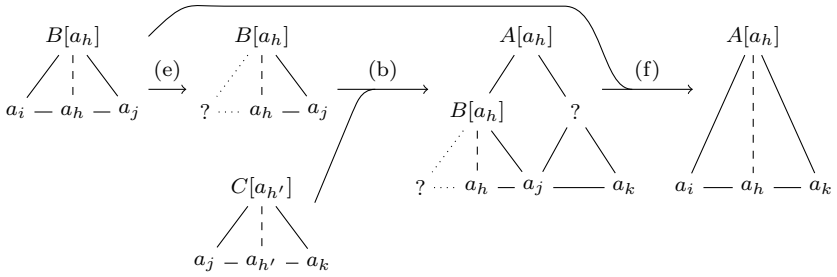

Figure 7: Illustration of the use of inference rules (e), (b) and (f) of bilexical recognition.

technical concepts, but has no linguistic pretences.

S	→	NP VP	(1)
NP	→	N	(0.4)
NP	→	PosPr N	(0.3)
NP	→	AP N	(0.2)
NP	→	VP	(0.1)
N	→	company	(0.6)
N	→	workers	(0.4)
PosPr	→	our	(1)
VP	→	Aux VP	(0.3)
VP	→	V NP	(0.7)
Aux	→	is	(1)
V	→	training	(0.9)
V	→	is	(0.1)
AP	→	training	(1)

In the above grammar, the probabilities of rules with a given nonterminal in the left-hand side always sum to 1. A probabilistic grammar for which this holds is called **proper**. The condition of properness is strongly related to the fact that context-free grammars are a **generative formalism**, that is, a grammar defines a set of objects by offering a number of operations to step-wise turn (partial) objects into larger objects. In the case of probabilistic context-free grammars that are proper, the range of applicable operations at each step of a left-most derivation forms a probability distribution.

With the grammar as above, the most probable parse of 'our company is training workers' is that of Figure 1(a). Its probability is the product of the

probabilities of all applied rules (or more precisely, rule occurrences). This is $1 \cdot 0.3 \cdot 1 \cdot 0.6 \cdot 0.3 \cdot 1 \cdot 0.7 \cdot 0.9 \cdot 0.4 \cdot 0.4 = 0.0054432$.

With a probabilistic input grammar $\mathcal{G}$, the intersection grammar $\mathcal{G}_\cap$ constructed for a given input string w is also probabilistic. A rule of the form $[i_0, A, i_m] \to [i_0, X_1, i_1] \cdots [i_{m-1}, X_m, i_m]$ is assigned the same probability as the rule $A \to X_1 \cdots X_m$ in the input grammar $\mathcal{G}$. Rules of the form $[i-1, a_i, i] \to a_i$ are assigned the probability 1.

The intersection grammar is not proper in general, but it has the attractive feature that parses of w in $\mathcal{G}$ have the same probability as the corresponding parses in $\mathcal{G}_\cap$. This means that the problem of finding the most probable parse of w in $\mathcal{G}$ can be translated to the problem of finding the most probable parse in $\mathcal{G}_\cap$. The latter problem is simpler than the original problem, as the input string no longer needs to be considered explicitly. In fact, we investigate this problem below for an arbitrary probabilistic context-free grammar $\mathcal{G}$, which may or may not be the result of intersection.

The algorithm in Figure 8 is a special case of an algorithm by Knuth, which generalizes Dijkstra's algorithm to compute the shortest path in a weighted graph. It finds the probability $p_{max}(A)$ of the most probable subparse with root labelled A. The value $p_{max}(S)$, where S is the start symbol, then gives us the probability of the most probable parse. The algorithm can be easily extended to return the parse itself. For notational convenience, we let $p_{max}(a) = 1$ for each terminal a. The set of terminals is here denoted as Σ.

In each iteration, the value of $p_{max}(A)$ is established for a nonterminal A; the set $\mathcal{E}$ contains all grammar symbols X for which $p_{max}(X)$ has already been established. Initially, this is Σ, as we let $p_{max}(a) = 1$ for each $a \in \Sigma$. The set $\mathcal{F}$ contains those nonterminals not yet in $\mathcal{E}$ that are candidates to be added next. Each nonterminal A in $\mathcal{F}$ is such that a subparse with root labelled A exists consisting of a rule $A \to X_1 \cdots X_m$, and subparses with roots labelled $X_1, \ldots, X_m$ matching the values of $p_{max}(X_1)$, $\ldots$, $p_{max}(X_m)$ found earlier. The nonterminal A for which such a subparse has the highest probability is then added to $\mathcal{E}$.

In a practical implementation, $\mathcal{F}$ and q would not be constructed anew for each iteration. They would merely be revised every time a nonterminal A is added to $\mathcal{E}$. This revision consists in removing A from $\mathcal{F}$ and finding new rules whose right-hand side nonterminals are now all in $\mathcal{E}$. This allows adding new elements to $\mathcal{F}$ and/or updating q to assign higher values to elements in $\mathcal{F}$. Typically, $\mathcal{F}$ would be organized as a priority queue.

Knuth's algorithm can be combined with construction of the intersection grammar. If the grammar has no cycles, several simplified algorithms exist.

MostProbableParse($\mathcal{G}$)

$\mathcal{E} \leftarrow \Sigma$
repeat
  $\mathcal{F} \leftarrow \{A \mid A \notin \mathcal{E} \wedge \exists A \rightarrow X_1 \cdots X_m [X_1, \ldots, X_m \in \mathcal{E}]\}$
  if $\mathcal{F} \leftarrow \emptyset$ **then**
    report failure and halt
  for all $A \in \mathcal{F}$ **do**
    $q(A) \leftarrow \max_{\substack{\pi=(A \rightarrow X_1 \cdots X_m): \\ X_1,\ldots,X_m \in \mathcal{E}}} p(\pi) \cdot p_{max}(X_1) \cdot \ldots \cdot p_{max}(X_m)$
  choose $A \in \mathcal{F}$ such that $q(A)$ is maximal
  $p_{max}(A) \leftarrow q(A)$
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{A\}$
until $S \in \mathcal{E}$
output $p_{max}(S)$

Figure 8: Knuth's generalization of Dijkstra's algorithm, applied to finding the most probable parse in a probabilistic context-free grammar $\mathcal{G}$.

A particularly simple algorithm is an extended form of CYK parsing for grammars in Chomsky normal form, which computes values $p_{max}([i, A, j])$ after computing all values $p_{max}([i', A', j'])$ with $i < i' < j' \leq j$ or $i \leq i' < j' < j$.

The probability of an ambiguous string is defined as the sum of the probabilities of all parses of that string. In contrast to finding the most probable derivation, the problem of finding the most probable string is generally difficult. The decision version of this problem is NP-complete if there is a specified bound on the string length, and undecidable otherwise.

Probabilistic parsing is particularly effective for lexicalized grammars, as it allows fine-grained encoding of dependencies between lexical elements. For example, the rule S[training] $\rightarrow$ NP[company] VP[training] could be given a high probability, whereas S[training] $\rightarrow$ NP[problem] VP[training] is given a low probability.

Probabilistic context-free grammars for natural languages are normally induced on the basis of samples of language use, rather than explicitly written by people. The simplest algorithms of grammar induction rely on input consisting of a multiset of parses, also known as a **tree bank**. Tree banks are often the result of manual, or (partially) automated annotation of a number of texts, for example newspaper articles.

The nonterminal label of a node in the tree bank together with the label of its childeren forms a rule. The probability of such a rule, say $A \rightarrow \alpha$, can be estimated as:

$$\frac{C(A \rightarrow \alpha)}{C(A)}, \tag{1}$$

where $C(A)$ is the number of nodes in the input data with label A and $C(A \rightarrow \alpha)$ is the number of occurrences of rule $A \rightarrow \alpha$ in the tree bank.

Probabilities of lexicalized rules are often computed as the product of probabilities of a number of features that together determine the rule. For example, the probability of $A[b] \rightarrow B[b]\ C[c]$ can be expressed as the probability of B given A and b, times the probability of C as second member in the right-hand side, given A, b and B, times the probability of c given A, b, B and C as second member. The last probability can be approximated as, for example, the probability of c given b and C. Such approximations lead to a reduced number of parameters, which can be estimated more accurately if the available tree bank is small.

6 Translation

Automatic translation between natural languages is one of the most challenging applications in NLP. State-of-the-art approaches to this task are based on syntactic models, usually enriched with statistical parameters. In this section we consider one such model, called **synchronous context-free grammar** (SCFG), which is a notational variant of the syntax-directed translation schemata originally developed in the theory of compilers (Aho and Ullman, 1972).

A SCFG consists of **synchronous rules**, each obtained by pairing two CFG rules with the same left-hand side. The right-hand sides of such a pair of CFG rules must consist of identical multisets of nonterminals, possibly ordered differently, and possibly combined with different terminal symbols. Furthermore, there is an explicit bijection that pairs occurrences of identical nonterminals in the two right-hand sides.

As an example, the synchronous rule $\langle \mathrm{VP} \rightarrow \mathrm{VB}^{\boxed{1}}\ \mathrm{PP}^{\boxed{2}},\ \ \mathrm{VP} \rightarrow \mathrm{PP}^{\boxed{2}}\ \mathrm{VB}^{\boxed{1}}\ \mathrm{ga} \rangle$ states that an English verb phrase composed of the two constituents VB ('verb in base form') and PP ('prepositional phrase') can be translated into Japanese by swapping the order of the translations of these constituents and by inserting the word 'ga' at the right. Note the use of integers within boxes as superscripts to indicate a bijection between nonterminal occurrences in the two context-free

rules.

A SCFG can derive pairs of sentences as follows. Starting with the pair of nonterminals $\langle S^{\boxed{1}},\ S^{\boxed{1}}\rangle$, synchronous rules are applied to rewrite pairs of nonterminals that have the same index. At the application of a rule, the indices in the newly added nonterminals are consistently renamed, in order to avoid clashes with the indices introduced at previous rewriting steps. The rewriting stops when all nonterminals have been rewritten.

The following toy SCFG will be the running example of this section:

$$\begin{array}{lllll}
s_1: & \langle & S \to A^{\boxed{1}}C^{\boxed{2}}, & S \to A^{\boxed{1}}C^{\boxed{2}} & \rangle \\
s_2: & \langle & C \to B^{\boxed{1}}S^{\boxed{2}}, & C \to B^{\boxed{1}}S^{\boxed{2}} & \rangle \\
s_3: & \langle & C \to B^{\boxed{1}}S^{\boxed{2}}, & C \to S^{\boxed{2}}B^{\boxed{1}} & \rangle \\
s_4: & \langle & C \to B^{\boxed{1}}, & C \to B^{\boxed{1}} & \rangle \\
s_5: & \langle & A \to a_1, & A \to a_2 & \rangle \\
s_6: & \langle & A \to a_1, & A \to \varepsilon & \rangle \\
s_7: & \langle & B \to b_1, & B \to b_2 & \rangle
\end{array}$$

In the represented translation, nonterminals B and S can be optionally inverted (rule s_2 or rule s_3), and the terminal symbol a_2, which is the translation of a_1 (by rule s_5), can be optionally deleted (rule s_6).

An example derivation of the string pair $\langle a_1b_1a_1b_1,\ a_2b_2b_2\rangle$ by the above SCFG is:

$$\begin{array}{rll}
\langle S^{\boxed{1}},\ S^{\boxed{1}}\rangle & \Rightarrow^{s_1} & \langle A^{\boxed{2}}C^{\boxed{3}},\ A^{\boxed{2}}C^{\boxed{3}}\rangle \\
& \Rightarrow^{s_3} & \langle A^{\boxed{2}}B^{\boxed{4}}S^{\boxed{5}},\ A^{\boxed{2}}S^{\boxed{5}}B^{\boxed{4}}\rangle \\
& \Rightarrow^{s_1} & \langle A^{\boxed{2}}B^{\boxed{4}}A^{\boxed{6}}C^{\boxed{7}},\ A^{\boxed{2}}A^{\boxed{6}}C^{\boxed{7}}B^{\boxed{4}}\rangle \\
& \Rightarrow^{s_4} & \langle A^{\boxed{2}}B^{\boxed{4}}A^{\boxed{6}}B^{\boxed{8}},\ A^{\boxed{2}}A^{\boxed{6}}B^{\boxed{8}}B^{\boxed{4}}\rangle \\
& \Rightarrow^{s_5} & \langle a_1B^{\boxed{4}}A^{\boxed{6}}B^{\boxed{8}},\ a_2A^{\boxed{6}}B^{\boxed{8}}B^{\boxed{4}}\rangle \\
& \Rightarrow^{s_7} & \langle a_1b_1A^{\boxed{6}}B^{\boxed{8}},\ a_2A^{\boxed{6}}B^{\boxed{8}}b_2\rangle \\
& \Rightarrow^{s_6} & \langle a_1b_1a_1B^{\boxed{8}},\ a_2B^{\boxed{8}}b_2\rangle \\
& \Rightarrow^{s_7} & \langle a_1b_1a_1b_1,\ a_2b_2b_2\rangle.
\end{array}$$

In the same way as a derivation in a CFG can be associated with a parse tree, a derivation in a SCFG can be associated with a pair of parse trees. These trees differ only in the (terminal) labels of the leaf nodes and in the ordering of siblings, as illustrated by Figure 9. We will refer to the two trees in a pair as the input tree and the output tree.

Given a SCFG $\mathcal{G}$ and a string w, the expression $w \circ \mathcal{G}$ denotes the set of all pairs of parse trees associated with derivations in $\mathcal{G}$ whose input tree has

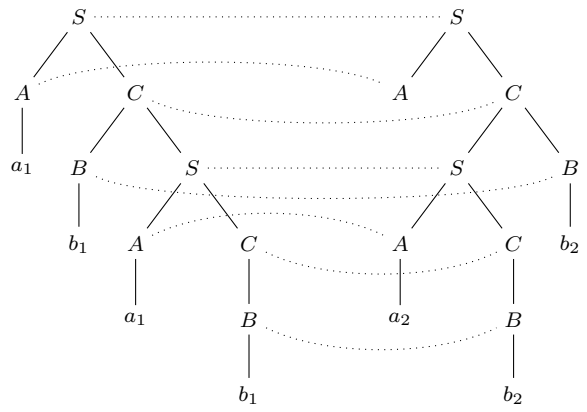

Figure 9: A pair of trees associated with a derivation in a SCFG. The dotted lines link pairs of nonterminal occurrences that had the same index during the rewriting process.

yield w. Note that all the strings that are translations of w under $\mathcal{G}$ can be easily enumerated if we can enumerate the elements of $w \circ \mathcal{G}$.

The set $w \circ \mathcal{G}$ can have size exponential in $|w|$, and the number of possible translations of w under $\mathcal{G}$ can likewise be exponential. (There may even be an infinite number of translations if there are synchronous rules whose left components are epsilon rules or unit rules.) In Section 3 we discussed a compact representation of a large set of parse trees, in the form of a CFG. Below, this is extended to a construction of a SCFG $\mathcal{G}'$ that represents $w \circ \mathcal{G}$ in a compact way. This construction is called **left composition**, and in full generality it can be applied to a finite automaton and a SCFG. As in Section 3, we simplify the discussion by assuming a special type of finite automaton that recognizes just one string $w = a_1 \cdots a_n$.

We further assume, without loss of generality, that synchronous rules from $\mathcal{G}$ are either of the form $\langle A \to \alpha,\ \ A \to \alpha' \rangle$, where α and α' are nonempty strings of indexed nonterminals, or of the form $\langle A \to x,\ A \to y \rangle$, where x and y can each be a terminal symbol or the empty string. In the former case, we use a permutation π to denote the bijective relation pairing the nonterminal occurrences in α and α', and write the synchronous rule as $\langle A \to B_1^{\boxed{1}} \cdots B_m^{\boxed{m}},\ A \to B_{\pi(1)}^{\boxed{\pi(1)}} \cdots B_{\pi(m)}^{\boxed{\pi(m)}} \rangle$.

The nonterminals of $\mathcal{G}'$ have the form $[i, A, j]$, where i and j denote input positions within w, and A is a nonterminal from $\mathcal{G}$.

LeftComposition$(w, \mathcal{G})$ {let $a_1 \cdots a_n = w$}

1: $\mathcal{G}' \leftarrow$ SCFG with start nonterminal $[0, S, n]$ and empty set of synchronous rules
2: **for all** $\langle A \rightarrow B_1^{\boxed{1}} \cdots B_m^{\boxed{m}},\ A \rightarrow B_{\pi(1)}^{\boxed{\pi(1)}} \cdots B_{\pi(m)}^{\boxed{\pi(m)}} \rangle$ from $\mathcal{G}$ **do**
3: **for all** $i_0, \ldots, i_m$ $(0 \leq i_0 \leq \ldots \leq i_m \leq n)$ **do**
4: add to $\mathcal{G}'$ the synchronous rule $\langle [i_0, A, i_m] \rightarrow [i_0, B_1, i_1]^{\boxed{1}} \cdots [i_{m-1}, B_m, i_m]^{\boxed{m}},\ [i_0, A, i_m] \rightarrow [i_{\pi(1)-1}, B_{\pi(1)}, i_{\pi(1)}]^{\boxed{\pi(1)}} \cdots [i_{\pi(m)-1}, B_{\pi(m)}, i_{\pi(m)}]^{\boxed{\pi(m)}} \rangle$
5: **for all** i $(1 \leq i \leq n)$ and $\langle A \rightarrow a_i,\ A \rightarrow y \rangle$ from $\mathcal{G}$ **do**
6: add to $\mathcal{G}'$ the synchronous rule $\langle [i-1, A, i] \rightarrow a_i,\ [i-1, A, i] \rightarrow y \rangle$
7: **for all** i $(0 \leq i \leq n)$ and $\langle A \rightarrow \varepsilon,\ A \rightarrow y \rangle$ from $\mathcal{G}$ **do**
8: add to $\mathcal{G}'$ the synchronous rule $\langle [i, A, i] \rightarrow \varepsilon,\ [i, A, i] \rightarrow y \rangle$
9: **return** $\mathcal{G}'$

This construction may introduce many nonterminals into $\mathcal{G}'$ that are useless in the same way as algorithm Intersection from Section 3 introduces useless nonterminals. Available techniques to eliminate useless nonterminals from $\mathcal{G}'$ are very similar to techniques discussed before.

If we remove the left components from synchronous rules of $\mathcal{G}'$, then we obtain a CFG $\mathcal{G}''$ that generates parse trees for all possible translations of w under $\mathcal{G}$. These parse trees differ from the output trees in $w \circ \mathcal{G}$ only in the labels of nodes. In the former there are labels of the form $[i, A, j]$ where in the latter there are labels A.

Returning to the running example, consider the input string $w = a_1 b_1 a_1 b_1$. With the SCFG given above, this string can be translated into the five strings $a_2 b_2 a_2 b_2$, $a_2 a_2 b_2 b_2$, $a_2 b_2 b_2$, $b_2 a_2 b_2$, or $b_2 b_2$. There are eight pairs of trees in $w \circ \mathcal{G}$, as there are three derivations with output $a_2 b_2 b_2$, and two derivations with output $b_2 b_2$. After applying left composition and reduction

of the grammar, we obtain:

$$
\begin{array}{lll}
\langle & [0,S,4] \rightarrow [0,A,1]^{\boxed{1}}\ [1,C,4]^{\boxed{2}}, & [0,S,4] \rightarrow [0,A,1]^{\boxed{1}}\ [1,C,4]^{\boxed{2}} \quad \rangle \\
\langle & [1,C,4] \rightarrow [1,B,2]^{\boxed{1}}\ [2,S,4]^{\boxed{2}}, & [1,C,4] \rightarrow [1,B,2]^{\boxed{1}}\ [2,S,4]^{\boxed{2}} \quad \rangle \\
\langle & [1,C,4] \rightarrow [1,B,2]^{\boxed{1}}\ [2,S,4]^{\boxed{2}}, & [1,C,4] \rightarrow [2,S,4]^{\boxed{2}}\ [1,B,2]^{\boxed{1}} \quad \rangle \\
\langle & [2,S,4] \rightarrow [2,A,3]^{\boxed{1}}\ [3,C,4]^{\boxed{2}}, & [2,S,4] \rightarrow [2,A,3]^{\boxed{1}}\ [3,C,4]^{\boxed{2}} \quad \rangle \\
\langle & [3,C,4] \rightarrow [3,B,4]^{\boxed{1}}, & [3,C,4] \rightarrow [3,B,4]^{\boxed{1}} \quad \rangle \\
\langle & [0,A,1] \rightarrow a_1, & [0,A,1] \rightarrow a_2 \quad \rangle \\
\langle & [0,A,1] \rightarrow a_1, & [0,A,1] \rightarrow \varepsilon \quad \rangle \\
\langle & [1,B,2] \rightarrow b_1, & [1,B,2] \rightarrow b_2 \quad \rangle \\
\langle & [2,A,3] \rightarrow a_1, & [2,A,3] \rightarrow a_2 \quad \rangle \\
\langle & [2,A,3] \rightarrow a_1, & [2,A,3] \rightarrow \varepsilon \quad \rangle \\
\langle & [3,B,4] \rightarrow b_1, & [3,B,4] \rightarrow b_2 \quad \rangle
\end{array}
$$

The size of a SCFG $\mathcal{G}$, written as $|\mathcal{G}|$, is defined as the sum of the number of nonterminal and terminal occurrences in its synchronous rules. Let r be the length of the longest right-hand side of a context-free rule that is the input or output component of a synchronous rule. The time and space complexity of left composition are both $\mathcal{O}(|\mathcal{G}| \cdot n^{r+1})$, where n is the length of the input string. In many practical applications, it is possible to factorize synchronous rules in such a way that the parameter r is reduced to a small integer. In the general case however, a SCFG cannot be cast into an equivalent form with r bounded by some constant. This implies exponential behaviour in the worst case.

Next to left composition of SCFG $\mathcal{G}$ with input string w_1 there is **right composition** of $\mathcal{G}$ with output string w_2. The definition of right composition is the natural mirror image of that of left composition, and it results in a SCFG $\mathcal{G}'$ that represents a set of tree pairs denoted by $\mathcal{G} \circ w_2$.

Left and right composition may be combined, one after the other in either order, to construct a SCFG representing a set of tree pairs $w_1 \circ \mathcal{G} \circ w_2$, that is, the set of all derivations of $\mathcal{G}$ with input w_1 and output w_2. Important applications include inducing machine translation components from text that was manually translated.

7 Further information

Formal properties of context-free grammars are discussed, amongst others, by Hopcroft and Ullman (1979) and Sippu and Soisalon-Soininen (1988). Interesting observations about the intersection of context-free languages and

regular languages, as relating to the distinction between parsing and recognition, are due to Lang (1994), which offers a different perspective from that of, for example, Ruzzo (1979).

The algorithm GeneratingIntersection is close to a bottom-up variant of the parsing algorithm by Graham et al. (1980). The presentation of recognition algorithms as deduction systems is commonly identified with Shieber et al. (1995) and Sikkel (1997). The underlying idea however can be traced back to Cook (1970). The time complexity of deduction systems, implemented as dynamic programming algorithms, was discussed by McAllester (2002).

The algorithm in Figure 6 has been adapted from Eisner and Satta (1999).

The problem of finding the most probable derivation is discussed by Knuth (1977) and Nederhof (2003) in the general case, and by Jelinek et al. (1992) for grammars in Chomsky normal form. The problem of finding the most probable string is discussed by Paz (1971), Casacuberta and de la Higuera (2000), Sima'an (2002) and Blondel and Canterini (2003).

The result that general SCFGs cannot be cast in a normal form with a bound on rule length is from Aho and Ullman (1969). NP-hardness of problems relating to translation are discussed in Satta and Peserico (2005).

The running example from the beginning of this chapter is adapted from Manning and Schütze (1999), which is recommended as a good introduction to statistical natural language processing. A good general textbook on NLP is Jurafsky and Martin (2000).

The main journal of NLP is **Computational Linguistics**. Of at least equal importance are several annual and biennial conferences, among which are **ACL** ('Annual Meeting of the Association for Computational Linguistics'), **EACL** ('European Chapter of the ACL'), **NAACL** ('North American Chapter of the ACL'), **COLING** ('International Conference on Computational Linguistics'), **EMNLP** ('Empirical Methods in Natural Language Processing') and **HLT** ('Human Language Technology').

8 Defining terms

Ambiguity Existence of more than one interpretation of an element of language, for example existence of several parses of one sentence, or several possible meanings of a word.

Disambiguation The process of identifying one preferred interpretation from a set of interpretations of an ambiguous element of language.

Formal language Language that is defined with mathematical rigour.

Grammar induction Acquiring a grammar out of a sample of language.

(Parse) item Element stored in the table of a parsing or recognition algorithm, representing existence of certain subparses.

Linguistics The study of natural language.

Machine translation Automated translation between natural languages.

Natural language Language used in human communication that evolved without too many deliberate considerations.

Natural language processing Automated analysis, generation or translation of language.

Parse (tree) Structural interpretation of a sentence.

Parse forest Structure containing a number of parses of one sentence.

Syntax The study of the structure of sentences, as composed of words.

Tree bank Multiset of parses, representing an annotation of a sample of language use.

References

Aho, A. and Ullman, J. (1969). Properties of syntax directed translations. *Journal of Computer and System Sciences*, 3:319–334.

Aho, A. and Ullman, J. (1972). *Parsing*, volume 1 of *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, N.J.

Bar-Hillel, Y., Perles, M., and Shamir, E. (1964). On formal properties of simple phrase structure grammars. In Bar-Hillel, Y., editor, *Language and Information: Selected Essays on their Theory and Application*, chapter 9, pages 116–150. Addison-Wesley, Reading, Massachusetts.

Blondel, V. and Canterini, C. (2003). Undecidable problems for probabilistic automata of fixed dimension. *Theory of Computing systems*, 36:231–245.

Casacuberta, F. and de la Higuera, C. (2000). Computational complexity of problems on probabilistic grammars and transducers. In Oliveira, A., editor, *Grammatical Inference: Algorithms and Applications*, volume 1891 of *Lecture Notes in Artificial Intelligence*, pages 15–24. Springer-Verlag.

Cook, S. (1970). Path systems and language recognition. In *ACM Symposium on Theory of Computing*, pages 70–72.

Eisner, J. and Satta, G. (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. In *37th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 457–464, Maryland, USA.

Graham, S., Harrison, M., and Ruzzo, W. (1980). An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2:415–462.

Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Jelinek, F., Lafferty, J., and Mercer, R. (1992). Basic methods of probabilistic context free grammars. In Laface, P. and De Mori, R., editors, *Speech Recognition and Understanding — Recent Advances, Trends and Applications*, pages 345–360. Springer-Verlag.

Jurafsky, D. and Martin, J. (2000). *Speech and Language Processing*. Prentice-Hall.

Knuth, D. (1977). A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5.

Lang, B. (1994). Recognition can be harder than parsing. *Computational Intelligence*, 10(4):486–494.

Manning, C. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

McAllester, D. (2002). On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537.

Nederhof, M.-J. (2003). Weighted deductive parsing and Knuth's algorithm. *Computational Linguistics*, 29(1):135–143.

Paz, A. (1971). *Introduction to Probabilistic Automata*. Academic Press, New York.

Ruzzo, W. (1979). On the complexity of general context-free language parsing and recognition. In *Automata, Languages and Programming, Sixth Colloquium*, volume 71 of *Lecture Notes in Computer Science*, pages 489–497, Graz. Springer-Verlag.

Satta, G. and Peserico, E. (2005). Some computational complexity results for synchronous context-free grammars. In *Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 803–810.

Shieber, S., Schabes, Y., and Pereira, F. (1995). Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24:3–36.

Sikkel, K. (1997). *Parsing Schemata*. Springer-Verlag.

Sima'an, K. (2002). Computational complexity of probabilistic disambiguation. *Grammars*, 5:125–151.

Sippu, S. and Soisalon-Soininen, E. (1988). *Parsing Theory, Vol. I: Languages and Parsing*, volume 15 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag.

Tomita, M. (1986). *Efficient Parsing for Natural Language*. Kluwer Academic Publishers.